UNIVERSIDAD DE PUERTO RICO

RECINTO UNIVERSITARIO DE MAYAGÜEZ

COMP 4036 – 070

PL Project – MAPL

Integrantes:

Aramis E. Matos Nieves

Lenier Gerena Meléndez

Yadiel Hernán Vélez Vargas

Introduction:

In this project, we aim to make a program that can achieve mainly two objectives: to have an easy to understand and easy to use grammar for both declaring and computing operations on a matrix or vector, and to balance the simplicity with concise and accurate calculations without needing to specify more information than is needed. To achieve this, we made the scope of our project reach only as far as simplicity and utility can lead us, which is simple algebraic computations. These computations range from simple sum and subtraction, to multiplications (scalar multiplication, cross product, dot product), transpose of a matrix, determinant, inverse of a matrix and adjugate of matrices. Much detail will be provided as to how the program works and how to use it to compute any of the previously mentioned algebraic expressions.

Language Tutorial:

To properly use the program, a set of rules must be followed in order to fully take advantage of the program's simplicity and linearity. Such rules are as dictated:

- The use of Capital letters are reserved for function purposes, such as PRINT, ADJUGATE, T (Transpose), and INV (Inverse).
    - This means that declaring a variable with a capital letter will result in a parse error, as well as any subsequent letters in the identifier's name (examples: matrix**C**alc, **V**ector, m**V**ar**R**esult.)
- Numbers are not considered as a valid string for the name of a variable, resulting in a parse error if attempted.
    - examples: matrix**0**, **1**stmatrix, **1**operation**2**result
- Upon entering a valid identifier, we proceed to assign it with either a vector or a matrix, this is preceded by a "=" sign before enclosing the matrix/vector in hard brackets "[]".
- If we wish to declare a vector, inside these hard brackets we place the necessary elements that are needed. If the amount of elements exceeds 1, just continue adding numbers with spaces between each other.
    - example: v = [1], v = [1 2 3 4 5]
- If we wish to declare a matrix, we must add an additional hard bracket for each row vector we wish to assign inside the matrix. In addition, we need to add a semicolon ";" each time we want to declare another row vector inside our matrix. Note that there can't be any space between the hard bracket that encloses the previous row vector, the semicolon, and the opening hard bracket for the next row vector inside our matrix.
    - example: m = [[1 2 3];[4 5 6]]

- To execute sums, subtractions, and multiplications, we separate the matrices/vectors with spaces from the operation symbol (+, -, *, .). There is an exception for the cross product function which will be mentioned shortly after.
    - Example: m + m, m - m, m * m, m . m
- To execute other functions like the previously mentioned ones, there is no need to add any parenthesis beforehand.
    - Example: PRINT m, ADJ m, INV m
- For the case of the determinant of a matrix we simply enclose the matrix or variable containing the matrix between the “|” symbols without leaving any space between the first “|”, the matrix in the middle, and the last “|” symbol.
    - Example: |m|
- Do note that the functions for adjugate (ADJ), Determinant (|matrix|), and cross product between matrices (matrix * matrix) expect the matrix to be a squared matrix, which is a matrix that both the number of rows and columns are of the same amount/count.
    - If the matrix ***m*** is of size 2x3 (two rows and 3 columns), then adjugate and determinant won't be possible to compute, therefore, returning an error. However, if the same ***m*** matrix is multiplied by another matrix whose rows equal the number of columns of the ***m*** matrix, then the cross product will compute. Example: ***m*** is 2x3 and ***sm*** is 3xn (where n is any number that indicates the amount of columns for ***sm***), then ***m * sm*** will return a result.

Language Reference Module:

To get more in-depth information about how the operations are performed and how both Vectors and Matrices are handled, please refer to the documentation in our github page.

Language Development:

In the lexer of our program, there are two main components; the token declaration or list, and the token specification. The token declaration is the tuple in which it contains all labels for the lexer to receive as an input. This is also where regular expressions are defined (and only those who are compatible with the re module in Python), and all of the rules are defined with declarations that are pre-attached with tht character “t_” (as an indicator for it being a token). Once the lexer has been built with all of the regular expressions and declarations for each label, the lex.lex() function is used, and this is where the input data will be fed.

However, it's important to know how this information will be fed and, in our case that contains multiple operations which need to take a special type of precedence in order to have a valid output from the input given, we define such precedences in a tuple. This also means that the input will only take a single argument (pertaining to a certain type of value) and said input will have the necessary and correct grammar symbols for each rule defined within our grammar. Here we also define what an expression is and what an expression is, what assignments are, what will be returned if the symbols are not compatible with the defined vector, what is a vector and a matrix, and finally what operations the program is capable of computing. If the provided expression matches with either the regular expression for a vector or for a matrix, we pass the object into the respective constructor in either class. Whenever we reach that the expression given is either a vector or a matrix in our grammar and a certain symbol for operation is given; in the program, (+, -, *, ., scalar multiplication),

we then check after knowing what this symbol (or lack-thereof) refers to within our grammar and if said properties of the given expression are accomplished for that operation to compute.

The previously mentioned checkup for the validated expression is done within the classes that have been defined for each use case (the vector class and the matrix class). Once we pass our input through the constructor for either type, we also compute the desired operation with the symbol that was given. At this moment, the program will check if said operation is possible if there exists a constraint. The operations that have such constraints are: Inverse (matrix class), adjugate (matrix class), determinant (matrix class), and transpose (matrix). These all require for the matrix to be of the “squared matrix” nature, in which the number of rows equal the number of columns in said matrix given. If this is not accomplished, then the class method will return an error with the previously mentioned attribute not present in the matrix. Another constraint is that cross product (*) needs to have the left matrix’s number of columns be equal to the right matrix’s number of rows, otherwise an error will occur. Something to note is that both matrix type and vector type have almost identical functions, vectors having less than matrix class. The functions that they share are sum, subtraction, and scalar multiplication. The one difference between these two is that, besides from the obvious fact that the vectors dont have the functions for determinant, transpose, adjugate and inverse, the vector class has a different multiplication function which is known as the dot product of vectors. In the end, the grammar will pass the statement (which will match either a vector or matrix regular expression, and if not, a syntax error returns) through the class and a result, be it the computation or an error, will return from it.

For this program, we focused on developing on Linux and Mac operating systems, but our main IDE was VSCODE, here, we developed all of the external classes required to compute the necessary methods for each computation that was envisioned to be included and fell under our defined scope. A graphical user interface wasn't in our development plans, so even though we use Python 3 to work with our program, the user is still asked via the

command terminal to enter input and the output is presented via the same terminal. However, issues didn't arise developing in this IDE of choice, since we could link up to our github repository and easily update progress in the program or use the updated version from another group member. Also, importing defined classes made it fairly simple to have multiple programs for multiple parts of the program.

The test methodology used throughout the development of MAPL was mostly through the use of specific test cases on functionalities as the language functions were being developed.This was mostly done during the development of the vector and matrix classes since the context free grammar was not yet developed and thus testing with an example file was limited to non-existent. However, once the grammar for the language was developed, the test program *tester.py* was developed to read from *program_example.txt* and verify the correctness of the operations being done.

Conclusions

This project has really made us appreciate the hard work, time and dedication required to build a programming language, the things we use for our day to day work without a second thought for their construction or design.It also helped us gain experience writing in python, since we only started working with the language a semester ago. Moreover, the whole project really brought many fascinating elements of computer science we never really thought about, such as regular expressions, context-free grammar, etc. and it was fascinating to see how computer languages have so much in common with natural ones, in their theory and implementation. Overall, this was a great experience and we would do it again if given the chance.